Reinforcement Learning

AW

Lecture Overview

1. Idea Of Reinforcement Learning

2. Q-Learning

3. Using Deep Learning

Simple RL Setting: Multi-armed Bandits

- Imagine a gambler in a casino faced with 2 slot machines.
- Each trial costs the gambler \$1, but pays \$100 with some unknown (low) probability.
- The gambler suspects that one slot machine is better than the other.
- What would be the optimal strategy to play the slot machines, assuming that the gambler's suspicion is correct?

Stateless Model: Environment at every time-stamp is identical (although knowledge of *agent* improves).

- Playing both slot machines alternately helps the gambler learn about their payoff (over time).
  - However, it is wasteful *exploration*!
  - Gambler wants to *exploit* winner as soon as possible.
- Trade-off between exploration and exploitation $\Rightarrow$ Hallmark of reinforcement learning

Naïve Algorithm and ϵ-Greedy Strategy

Naïve:

Exploration: Play each slot machine for a fixed number of trials.

Exploitation: Play the winner forever

- Might require a large number of trials to robustly estimate the winner
- If we use too few trials, we might actually play the poorer

ϵ-Greedy:

- Probabilistically merge exploration and exploitation.
  - Play with probability ϵ
    - choose the trial when you play a random machine (probability ½),
  - in the rest of the trials (i.e. with probability $1 - \epsilon$) play the machine with highest current accumulated payoff.
- Main challenge in picking the proper value of $\epsilon \Rightarrow$ Decides trade-off between exploration and exploitation.
- Annealing: Start with large values of ϵ and reduce slowly.

Optimistic Gambler: Upper Bounding

- Upper-bounding represents optimism towards unseen machines ⇒ Encourages exploration.
- Empirically estimate mean probability of winning μ_i and standard error $\sigma_i = \sqrt{\frac{\mu_i(1-\mu_i)}{n_i}}$ of payoff of the i^{th} machine using its n_i trials. For standard confidence (e.g. $c = .95$) find margin of error $ME(\mu_i) = z_c^* \sigma_i$
- Pick the slot machine with largest value of mean plus margin of error $= \ \mu_i \ + ME(\mu_i)$
  - Note the $\sqrt{n_i}$ in the denominator, because it is *sample* standard deviation.
  - Rarely played slot machines more likely to be picked because of optimism.
  - Value of c decides trade-off between exploration and exploitation.

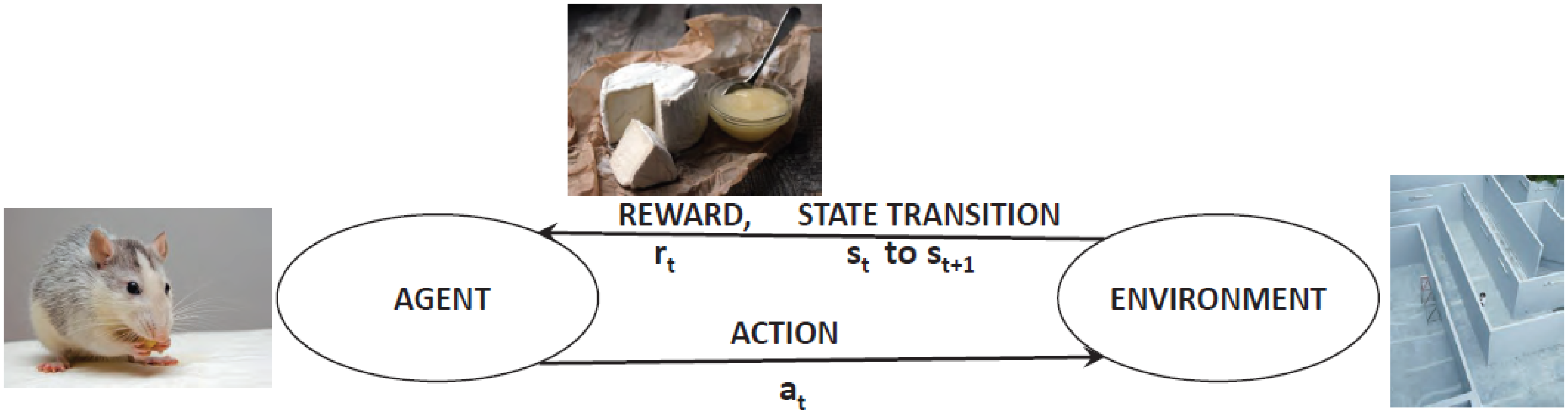
Multi-Armed Bandits vs Reinforcement Learning

- Multi-armed bandits is the simplest form of reinforcement learning.
  - The model is stateless, because the environment is identical at each time-stamp.
- Same action is optimal for each time-stamp.
  - Not true for classical reinforcement learning like Go, chess, robot locomotion, or video games.
  - State of the environment matters!
- Reinforcement Learning – Markov Decision Processes (MDP)
  - Environment has states
  - Agent has actions
  - Each action has reward/cost

MDP: Examples from Four Settings

- *Agent:* Mouse in the maze, chess player, gambler, robot
- *Environment:* maze, chess rules, slot machines, virtual test bed for robot
- *State:* Position in maze, chess board position, unchanged, robot joints
- *Actions:* Turn in maze, move in chess, pulling a slot machine, robot making step
- *Rewards:* cheese for mouse, winning chess game, payoff of slot machine, virtual robot reward
- MDP can be represented by its table where each entry is transition:
  - $Initial\ state, action, reward, final\ state$
  - Transitions may have probabilities assigned

Basic Framework of Reinforcement Learning



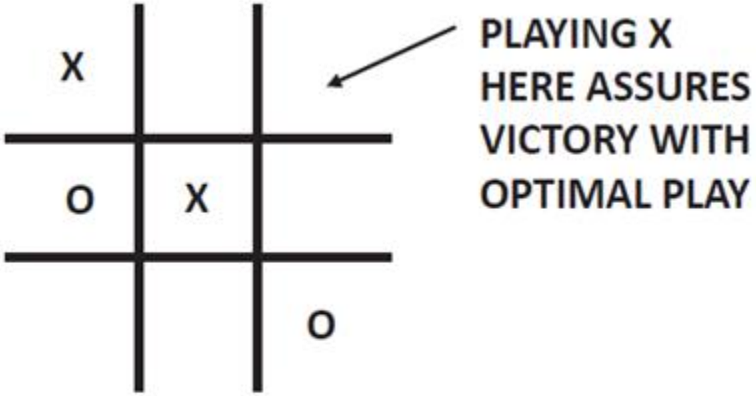
1. AGENT (MOUSE) TAKES AN ACTION a_t (LEFT TURN IN MAZE) FROM STATE (POSITION) s_t
2. ENVIRONMENT GIVES MOUSE REWARD r_t (CHEESE/NO CHEESE)
3. THE STATE OF AGENT IS CHANGED TO s_{t+1}
4. MOUSE'S NEURONS UPDATE SYNAPTIC WEIGHTS BASED ON WHETHER ACTION EARNED CHEESE

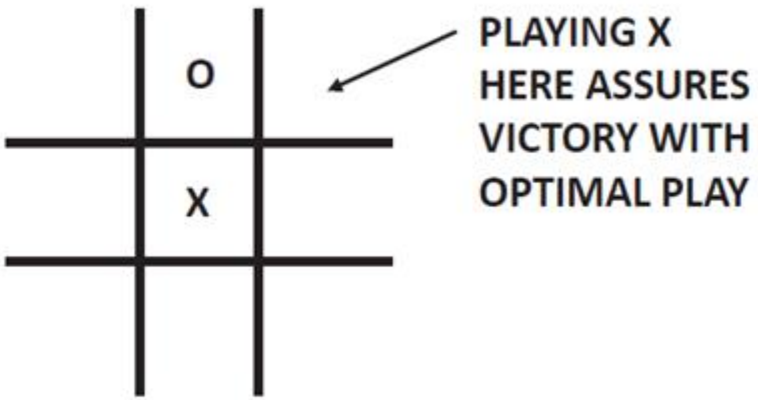
OVERALL: AGENT LEARNS OVER TIME TO TAKE STATE-SENSITIVE ACTIONS THAT EARN REWARDS

Role of Traditional Reinforcement Learning

- Traditional reinforcement learning: Learn through trial and-error the long-term value of each state.
- Long-term values are not the same as rewards.
  - Rewards not realized immediately because of stochasticity (e.g., slot machine) or delay (board game).

Example: *Game of tic-tac-toe, chess, or Go:* The state is the position of the board at any point, and the actions correspond to the moves made by the agent. The reward +1, 0, or −1 (win, draw, or loss), is *received at the end of the game*.





Learning for Tic-Tac-Toe and ϵ-greedy Algorithm

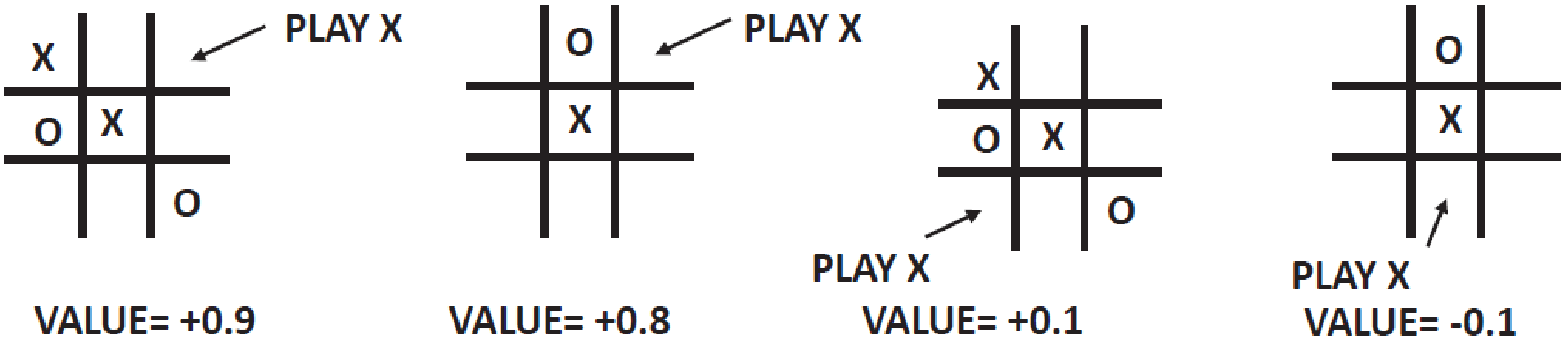
Assume that a fixed pool of humans is available as opponent to train the system (self-play possible).

- A move occurring r moves earlier than the game termination earns *discounted* rewards of $\{\gamma^{r-1}, 0, -\gamma^{r-1}\}$.
  - Future rewards would be less certain in a replay
- Maintain table of values of state-action pairs (for tic-tac-toe state is defined by value $\{o, x, -\}^9$, action is choosing position $1, \dots, 9$)
- Initialize to small random values.
  - In multi-armed bandits, we only had values on actions.
  - Table contains unnormalized total reward for each (state, action) pair $\Rightarrow$ Normalize to average reward.
- Use -greedy algorithm with *normalized* table values to simulate moves and create a game.

After game: Increment at most 9 entries in the unnormalized table with values from $\{\gamma^{r-1}, 0, -\gamma^{r-1}\}$ for r moves to termination and win/loss.

- Repeat the steps above.

Hypothetical Example: At the End of Training



- Typical examples of normalized values of moves
  - ϵ −greedy will learn the strategic values of moves.
- Rather than state-action-value triplets, we can equivalently learn state-value pairs.

How is RL Related to ANN?

- RL in Tic-Tac-Toe is glorified learning by memorization and repetition
- Works only for toy settings with few states.
  - Number of board positions in chess is huge.
  - Need to be able to *generalize* to unseen states.

Function Approximator: Rather than a table of (state, value) pairs, we can have a *neural network* that learns states-to-value maps.

- The *parameters* in the neural network are substitute for the table.

ϵ-Greedy ANN for Chess (Primitive - Don't Try It)

- Convolutional neural network takes board position as input and produces position value as output.
- Use -greedy algorithm on output values to simulate a full game.

After game of n moves: Create n training points with board position as input feature map and targets from $\{\gamma^{r-1,} 0, -\gamma^{r-1}\}$ depending on move number and win/loss.

- Update neural network with these n training points if the same position occurs again in a new game averaging rewards over games.
- Repeat the steps above.

Reinforcement Learning in Chess and Go

The reinforcement learning systems, *AlphaGo* and *Alpha Zero*, have been designed for chess, Go, and shogi.

- Combines various advanced deep learning methods and Monte Carlo tree search.
- Plays positionally and sometimes makes sacrifices (much like a human).
  - Neural network encodes evaluation function learned from trial and error.
  - More complex and subtle than hand-crafted evaluation functions by conventional chess software.
- Generalize to unseen states in training.
- Deep learner can recognize subtle positional factors because of trial-and-error experience with *feature engineering*.

More Challenges

- Chess and tic-tac-toe are *episodic*, with a maximum length to the game (9 for tic-tac-toe and ≈6000 for chess).
- The -greedy algorithm updates episode by episode.
- What about infinite Markov decision processes like robots or long episodes?
  - Rewards received continuously.
  - Not optimal to update episode-by-episode.

Solution: Value function and Q-function learning can update after each *step* with *Bellman's equations*.

Lecture Overview

1. Idea Of Reinforcement Learning

2. Q-Learning

3. Using Deep Learning

Infinite Markov Decision Process

- Continuous (with discrete time) process generates sequence of infinite length $s_0 a_0 r_0 s_1 a_1 r_1 s_2 \ldots s_t a_t r_t s_{t+1} \ldots$
- The cumulative reward R_t at time t is given by the discounted sum of the immediate rewards for $\gamma \in (0, 1)$:
- $R_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \gamma^3 \cdot r_{t+3} \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$
- Future rewards worth less than immediate rewards ($\gamma < 1$).
- Choosing $\gamma < 1$ is not essential for episodic processes but critical for long MDPs (otherwise sum diverges no matter 1-step reward).

Bootstrapping Intuition

- Consider a MDP in which we are predicting values (e.g., long-term rewards) at each time-stamp.
  - A partial simulation of the future can improve the prediction at the current time-stamp.
  - This improved prediction can be used as the ground-truth at the current time stamp.

Tic-tac-toe example: Parameterized evaluation function for board.

  - After our opponent plays the next move, and board evaluation changes unexpectedly, we go back and correct parameters.

Temporal difference learning:

- Use difference in prediction caused by partial lookaheads (treat it as error for purposes of update).

Chess Example (hypothetical)

- Why is the minimax evaluation over a game tree of a chess program at 10-ply (ply=move) stronger than that using the 1-ply board evaluation?
  - Because evaluation functions are imperfect (can be strengthened by "cheating" with data from future)!
  - If chess were solved (like checkers today), the evaluation function at any ply would be the same.
  - The minimax evaluation at 10 ply can be used as a "ground truth" for updating a parameterized evaluation function at current position!
- Samuel's checkers program was the pioneer (called *TD-Leaf* today)
- Variant of idea used by TD-Gammon, *Alpha Zero*.

Q-learning

- Instead of minimax over a tree, we can use one-step lookahead
- Let $Q(s_t, a_t)$ be a table containing optimal values of (state,action) pairs (best value of action a_t in state s_t).
  - Assume we play tic-tac-toe with ϵ-greedy algorithm and $Q(s_t, a_t)$ initialized to random values.
- Instead of random move (=Monte Carlo), make the following update: $Q(s_t, a_t) = \ r_t \ + \gamma \cdot \max_a Q(s_{t+1}, a)$
  - Or gentler update:

$$Q(s_t, a_t) = Q(s_t, a_t)(1 - \alpha) + \alpha \left(r_t \ + \gamma \cdot \max_a Q(s_{t+1}, a) \right)$$

This is Bellman equation – discounts forward are equivalent to discounts backward at a later state.

- Idea: reverse the walk from the end back when computing the reward to forward when computing actions

Intuition: Why Does it Work?

Tic-tac-toe example:

- Most of the updates we initially make are not meaningful in tic-tac-toe.
  - We started off with random values.
- However, the update of the value of a next-to-terminal state is informative.
- The next time the next-to-terminal state occurs on RHS of Bellman equation, the update of the next-to-penultimate state will be informative.
- Over time, we will converge to the proper values of all $(state, action)$ pairs.

Bellman Equations Formally

Given:

- Transition table T contains admissible transitions s, a, s' from state s to state s' when action a is taken (or probabilities of such transitions if a occurs in more than one transition from s)
- Policy $\pi: S \to A$ specifies action taken at state
- Reward of action $r: A \to R$
- Value function $V^\pi : S \to R$ specifies the value of following policy π starting in state s
- State-action value function $Q^\pi: S \times A \to R$ specifies the value of the reward starting in state s, taking action a, and then continuing according to policy π
- Recurrence for policy π is $Q^\pi(s,a) = r(a) + \gamma \cdot V^\pi(s')$ where $a = \pi(s)$ and (s, a, s') is a transition in transition table T
- Then $V^\pi(s) = Q\big(s, \pi(s)\big)$

Need optimal policy π^*. For must hold

Bellman Equations:

1. $Q^{\pi^*}(s,a) = r(a) + \gamma \cdot V^{\pi^*}(s') \quad \big(\text{or } r(a) + \sum_{(s,a,s')\in \mathrm{T}} p(s,a,s') V^{\pi^*}(s')\big)$
2. $V^{\pi^*}(s) = \max_a Q^{\pi^*}(s,a)$
3. $\pi^*(s) = \operatorname{argmax}_a Q^{\pi^*}(s,a)$

Value Iteration Algorithm

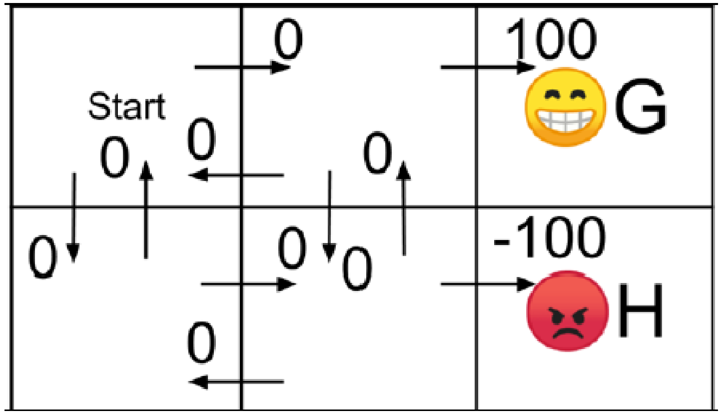
1. Start with for all s initialize $\mathrm{Q}_0^*(s, a) = 0$ (or random values – theoretically does not matter
2. For $i = 1, \ldots, n$
   a. Given Q_i^*, calculate for all states $s \in S$ and actions $a \in \mathrm{A}$:

$$Q_{i+1}(s, a) = r(a) + \gamma \cdot \max_{\substack{b \\ (s,a,s') \in \mathrm{T}}} Q_i(s', b)$$

The line 2.a is known as value update

Value Iteration Example

Game:



Initialization step:

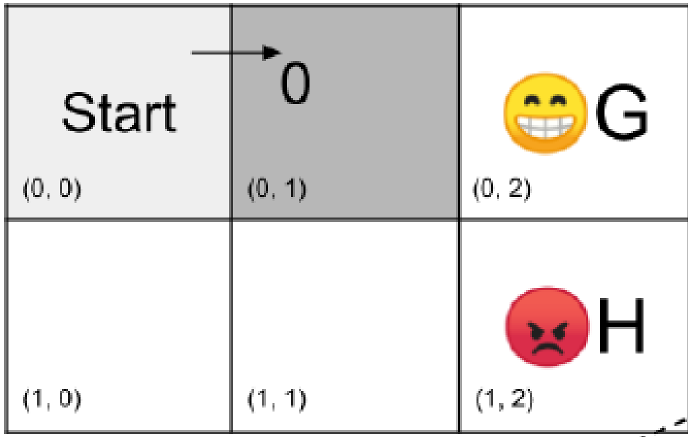
(0,0)	(0,1)	G (0,2)
(1,0)	(1,1)	H (1,2)

Environment

Action / State	←	↓	→	↑
(0,0)	0	0	0	0
(0,1)	0	0	0	0
(0,2)	0	0	0	0
(1,0)	0	0	0	0
(1,1)	0	0	0	0
(1,2)	0	0	0	0

Q Table

Value Iteration Example (cont.)



Mode: Exploration

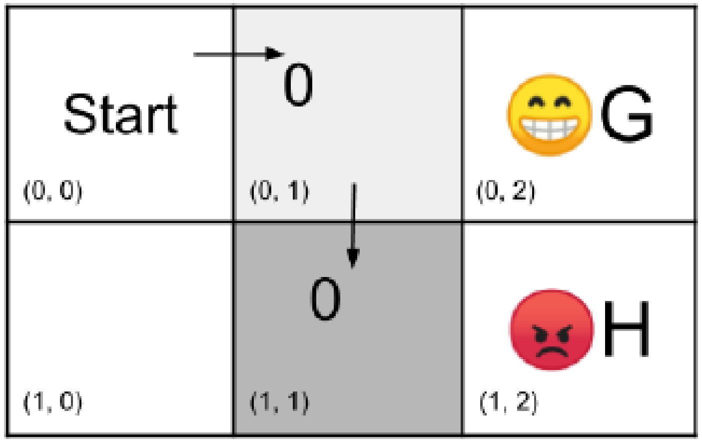
$$\underbrace{Q((0,0),\rightarrow)} = \text{reward} + \gamma * \underbrace{\max_{a'} Q((0,1), a')}$$

$$Q((0,0),\rightarrow) = 0 + 0.9 * \max(0, 0, 0, 0) = 0$$

State \ Action	←	↓	→	↑
(0,0)	0	0	0	0
(0,1)	0	0	0	0
(0,2)	0	0	0	0
(1,0)	0	0	0	0
(1,1)	0	0	0	0
(1,2)	0	0	0	0

Q Table

Value Iteration Example (cont.)



Mode: Exploration

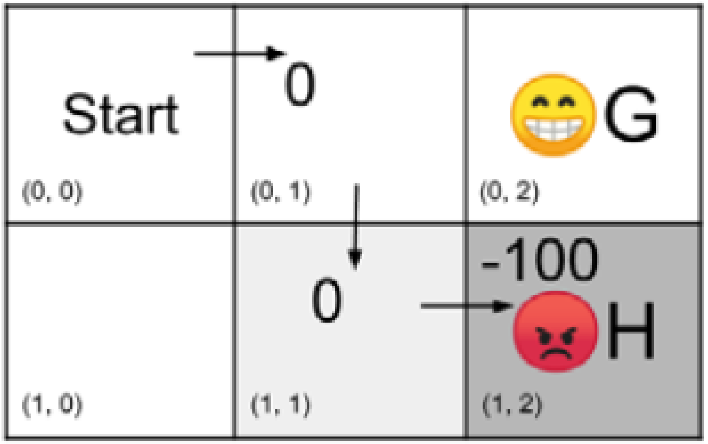
$Q((0,1),\downarrow) = \text{reward} + \gamma * \max_{a'} Q((1,1), a')$

$Q((0,1),\downarrow) = 0 + 0.9 * \max(0, 0, 0, 0) = 0$

State \ Action	←	↓	→	↑
(0,0)	0	0	0	0
(0,1)	0	0	0	0
(0,2)	0	0	0	0
(1,0)	0	0	0	0
(1,1)	0	0	0	0
(1,2)	0	0	0	0

Q Table

Value Iteration Example (cont.)



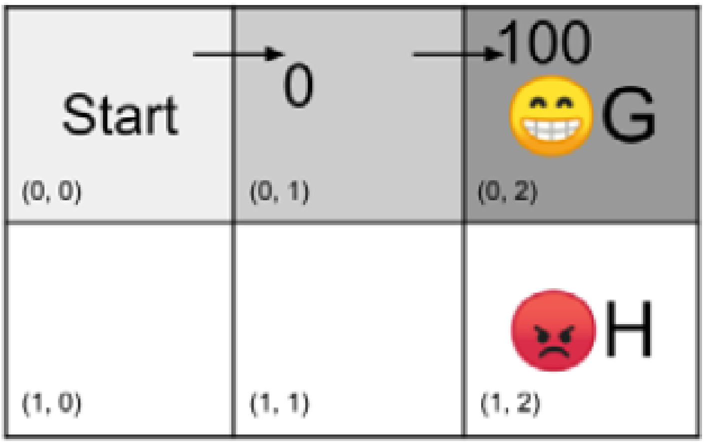
Mode: Exploration

Q((1,1),→) = reward = -100

Action / State	←	↓	→	↑
(0,0)	0	0	0	0
(0,1)	0	0	0	0
(0,2)	0	0	0	0
(1,0)	0	0	0	0
(1,1)	0	0	-100	0
(1,2)	0	0	0	0

Q Table

Value Iteration Example (cont.)



Mode: Exploration

$\underbrace{Q((0,1),\rightarrow)}$ = reward = 100

Action / State	←	↓	→	↑
(0,0)	0	0	0	0
(0,1)	0	0	100	0
(0,2)	0	0	0	0
(1,0)	0	0	0	0
(1,1)	0	0	-100	0
(1,2)	0	0	0	0

Q Table

SARSA Variant of Q-learning

SARSA: State-Action-Reward-State-Action

- $Q_t(s, a)$ be the value of action a in state s at time t when following the ϵ-greedy policy.
- Mix of exploration and estimation

Idea: an improved estimate of $Q_t(s, a)$ via bootstrapping is

$r(a) + \gamma Q_{t+1}(s', b)$ where $(s, a, s') \in T$

This follows from $R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i} = r_t + \gamma R_{t+1}$

SARSA: Instead of episodic update, we can update the table containing $Q_t(s, a)$ after performing a by ϵ-greedy, observing s' at time $t + 1$ and then computing new action b again using -greedy:

$Q_t(s, a) = r(a) + \gamma Q_{t+1}(s', b)$ where $(s, a, s') \in T$

On-Policy vs Off-Policy Learning

- SARSA: optimal reward in the next step is not used for computing updates. Next step is updated using the same ϵ-greedy policy to obtain the action a_{t+1} for computing the target values.
- SARSA: On-policy learning is useful when learning and inference cannot be separated.
  - A robot who continuously learns from the environment.
  - The robot must be cognizant that exploratory actions have a cost (e.g., walking at edge of cliff).
- Q-learning: Off-policy learning is useful when we don't need to perform exploratory component during inference time (have non-zero during training but set to 0 during inference).
  - Tic-tac-toe can be learned once using Q-learning, and then the model is fixed.

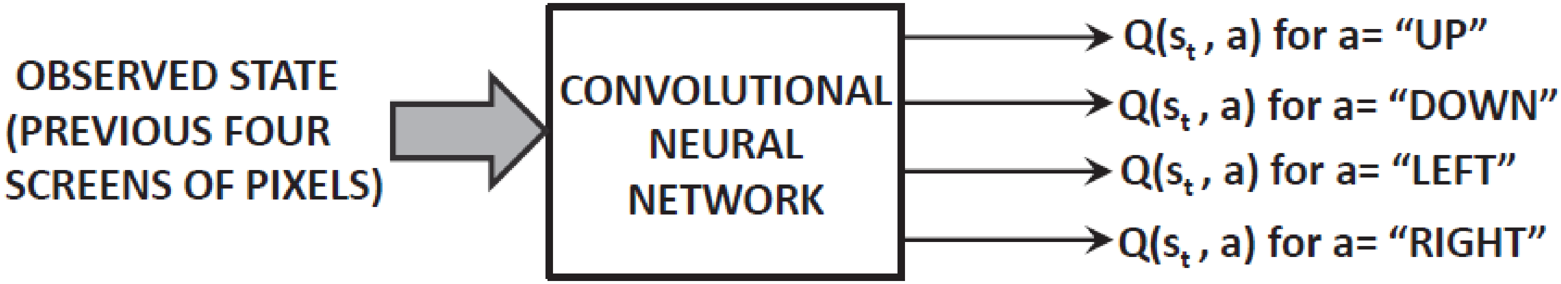
Lecture Overview

1. Idea Of Reinforcement Learning

2. Q-Learning

3. Using Deep Learning

The Main Approach

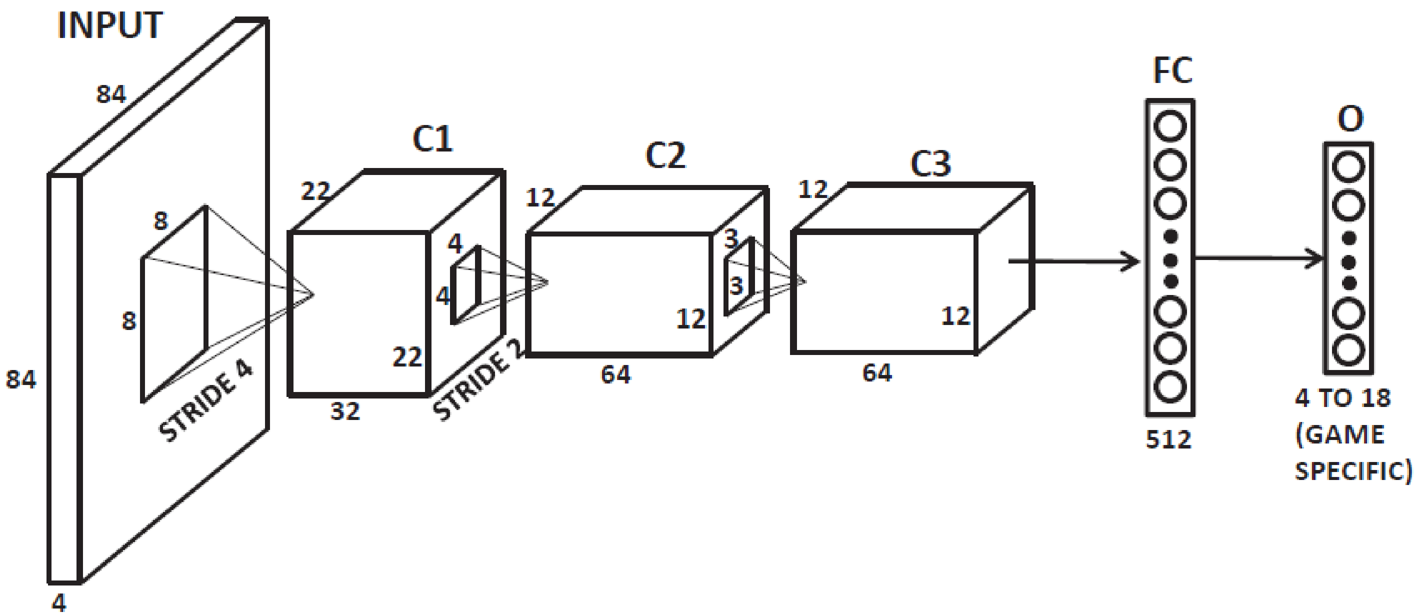


- When the number of states is large, the values $Q(s_t, a_t)$ are predicted from state s_t representation $\vec{x}_t$ rather than tabulated.

$$F(\vec{x}_t, W, a) \ = \hat{Q}(s_t, a)$$

- $\vec{x}_t$: Previous four screens of pixels in Atari

Specific Details of Convolutional Network



- Same architecture with minor variations was used for all Atari games.

Neural Network Updates for SARSA

- The neural network outputs $F(\vec{x}_t, W, a_t)$.
  - We must wait to observe state $\vec{x}_{t+1}$ and then set up a "ground-truth" value for the output using Bellman's equations:

$$\text{Bootstrapped Ground Truth} = r_t + \gamma \max_a F(\vec{x}_{t+1}, W, a)$$

$$\text{Loss: } L_t = (\underbrace{\left[r_t + \gamma \max_a F(\vec{x}_{t+1}, W, a) \right]}_{\text{Treat as constant ground–truth}} - F(\vec{x}_{t+1}, W, a))^2$$

Update:

$$W \Leftarrow W + \alpha(\underbrace{\left[r_t + \gamma \max_a F(\vec{x}_{t+1}, W, a) \right]}_{\text{Constant ground–truth}} - F(\vec{x}_t, W, a)) \frac{\partial F(\vec{x}_t, W, a_t)}{\partial W}$$

Neural Network Updates for Q-Learning

- The neural network outputs $F(\vec{x}_t, W, a\_t)$.
- We must wait to observe state $\vec{x}_{t+1}$ and simulate a_{t+1} with
- ϵ-greedy and then set up a "ground-truth" value:

$$\text{Bootstrapped Ground Truth} = r_t + \gamma F(\vec{x}_{t+1}, W, a_{t+1})$$

$$\text{Loss: } L_t = (\underbrace{[r_t + \gamma F(\vec{x}_{t+1}, W, a_{t+1})]}_{\text{Treat as constant ground-truth}} - F(\vec{x}_{t+1}, W, a_t))^2$$

Update:

$$W \Leftarrow W + \alpha(\underbrace{[r_t + \gamma F(\vec{x}_{t+1} + 1, W, a_{t+1})]}_{\text{Constant ground-truth}} - F(\vec{x}_t, W, a_t)) \frac{\partial F(\vec{x}_t, W, a_t)}{\partial W}$$

The Algorithm

- Initialize weights at random. Repeat the following at time-stamps $t = 1,2 \ldots,$ at which action a_t and reward r_t has been observed, to use the following training process for updating the weights W:

1. Perform a for ward pass through network to compute $\hat{Q}_{t+1} = \max_a F(\vec{x}_{t+1}, W, a)$ taking as input $\vec{x}_{t+1}$.
   - Note $\hat{Q}_{t+1} = 0$ if after performing a_t game ends. Must be so because $Q_t = r_t + \gamma\, \hat{Q}_{t+1}$ for observed action a_t at time t, so we create a *surrogate* for the target value at time *t* pretending that it is an observed value.
2. Perform a forward pass through the network with input $\vec{x}_t$ to compute $F(\vec{x}_t, W, a_t)$.
3. Compute a loss $L_t = \left(r_t + \gamma \hat{Q}_{t+1} - F(\vec{x}_t, W, a_t)\right)^2$. This loss is only for NN output node corresponding to a_t; the loss for other actions is 0.
4. Backpropagate the loss in the network with input $\vec{x}_t$. Update the weight vector W.

Reading

- Ch 9, sec 9.1-9.4.2.